Rit
— Embedded Rc in Text —

Kenji Arisawa, Aichi University
arisawa@aichi-u.ac.jp

ABSTRACT

Rit is a PHP like text processor that is designed for use in Plan 9. Embedded scripts are between “${” and “}” in text. In processing the text, Rit collaborates with Rc – the Plan 9 shell – using inter-process communication of Plan 9. This approach has three advantages: (1) we can use full functionality of Rc in the embedded area; (2) we can make Rit source small and simple; (3) Users need not learn new scripting language. This paper describes Rit v.1.5 with some examples.

1. Introduction

Writing Web documents has become one of the most popular style in publishing today. Many of these documents are based on HTML and Javascript. However some of them need help with server side scripting code.

Consider the case that we need to introduce some server side codes to an existing Web document. It will require much modification to rewrite the document in accordance with grammar of the scripting language. If the server side codes can be embedded in existing document, the amount of modification will be much reduced.

There are several text processors for embedded code. PHP[1] is one of the most popular ones. According to the document, PHP is a general-purpose scripting language that is especially suited for Web development and can be embedded into HTML. Rit[1] is similar to PHP in this respect. However the basic philosophy is much different.

PHP is a big software and provides new language for the embedded code. On the other contrary, Rit is small and simple. Rit does almost nothing to embedded script. Rit merely parses text to determine code areas; the areas are passed to Rc[2][3] so that they are interpreted and executed by Rc.

This approach has the following advantages:

- We can use full functionality of Rc,
- We can make Rit source small and simple.
- Users need not learn new scripting language,

The last advantage is important for users because users want to use their familiar tools.

[1] PHP http://jp.php.net/manual/en/introduction.php

Some people want to write in Perl and others, Python, etc. Thus, there exist text processors that handle embedded Perl(Mason[2], ePerl[3], etc), embedded Python(EmPy[4], Spyce[5], etc). These scripting languages are powerful enough to handle input by themselves. However Python and Perl are so big. Loading big executable is itself a disadvantage to use in Web services.

To the contrary, both Rit and Rc are small[6]. Rit just collaborates with Rc. Rc is well designed shell which enables users to use variety of tools if necessary. Thus, we can keep traditional developing style of UNIX or Plan 9: not standing on single big software, but standing on small neat tools.

Javascript is widely used in recent Web documents. Some of code that would have been written for server side can be written in Javascript. Thus, code needed on server side is getting smaller, which means, in many cases, we don't need big do-all software on server side.

2. History

Rit was developed in 2004. The first version 1.0 was released in late 2004. The name came from "*Rc In Text*". The syntax has been kept unchanged up to now. Reason of the update was mainly bug fix. However, there has been a problem: the grammar has not been defined in the document. Thus, the evaluation order and the semantics have been left ambiguous. The problem is fixed in version 1.4. Range option is added to version 1.5.

3. Rules

A single symbol "`$`" plays the role in Rit. With the "`$`", Rit has only five syntax rules.

- `${` *code* `}`
  executes Rc code. Multi-line is allowed in the code.

- `${` *code* `}$`
  is same as above but trims trailing *NL* (`'\n'`) in the output of last command. See "*Empty command*" and "*Multi-Lines*" sections both in the section "*Examples*" for the meaning of "last command", and see also "*Last command with no output*" in the section "*Known Bugs*".

- `$`*var*
  is equivalent to "`${echo -n $`*var*`}`". Here "*var*" is a sequence of alphas, numerics or underscores.

- `$`*NL*
  means newline escape.

---

[2] Mason http://www.masonhq.com/

[3] ePerl http://www.ossp.org/pkg/tool/eperl/

[4] EmPy http://www.alcyone.com/pyos/empy/

[5] Spyce http://spyce.sourceforge.net/

[6] Code sizes of Rit and Rc are 45KB(73KB) and 94KB(144KB) respectively. On the other hands, sizes of Python 2.4 and Perl 5.8 are 1.5MB(4.0MB) and 4.5MB(9.6MB) respectively. These values are on i386 PC. Values in parenthesis include symbol table. Sizes of Python and Perl depend on modules that are included in compilation.

- `$$...$`
  i,e., a sequence of n (> 1) dollars is reduced to $n - 1$ dollars. For example "`$$$`" is reduced to "`$$`".

Dollar "`$`" apart from above rules is shown as it is.

Rc comment continues up to end of line in accordance with Rc grammar. In parsing code area, Rit skips Rc string as well as "{ }" nest of Rc block.

An additional order rule is required to evaluate the effects of "`$`":

1. `}$`
2. `$$...$`
3. `$`*NL* , `${` , `$`*var*

which means the following text

```
${echo alice}$$$${echo bob}
```

produces the output

```
alice$${echo bob}
```

4. Internals

Consider the case that we have two or more code areas as illustrated in Fig.1. These areas must be processed by the same Rc so that commands in these areas can affect succeeding areas.

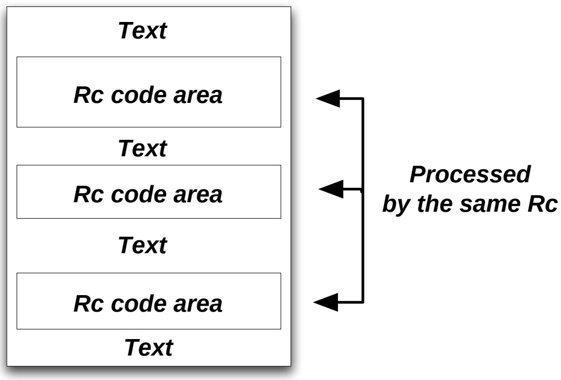


Fig.1: Structure of Rit text

Fig.2 shows how Rit text is processed by Rit and Rc.

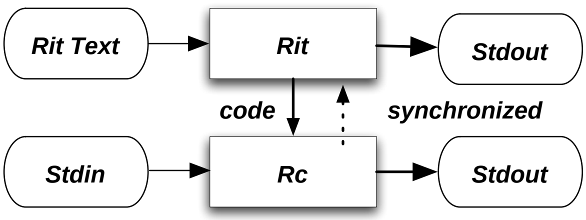


Fig.2: Data flow

In this figure, "*Rit Text*" is a text with embedded Rc code. "*Stdin*" and "*Stdout*" are standard input and standard output respectively. Data flow from "*Stdin*" to "*Rc*" comes from commands that read data from "*Stdin*". Communication channels between *Rit* and *Rc* are pipes: solid arrow is a named pipe and dashed arrow is a regular pipe. The former is used for passing code to Rc, and the latter, for synchronization.

5. Usage

5.1. Synopsis

`rit` [`-Dbes`] [*file* [*arg* ...]]

5.2. Description

Rit reads a file in the first argument of the command. Following the file "*args*" may be given. These *args* are passed to Rc so that Rc can get them as arguments. If file is not given, Rit reads Rit text from "*stdin*".
File name "`.`" is special. That means standard input. The "`.`" is provided so that Rit can pass arguments to embedded code.

Reading Rit text from "*stdin*" will not work if embedded code also read data from "*stdin*".

5.3. Options

- `-D` **:** debug
- `-b` **:** Without this option, "`$0`" is the path to a file. If this option is given, "`$0`" is base name of the path.
- `-e` **:** With this option, Rit exits immediately when Rit receives an error message from Rc.
- `-r` *range* **:** With this option, we can extract a part of Rit text from the file.
- `-s` **:** used with file so that the first line in the file is skipped.

Format of *range* is

begin`,`*end*

where "*begin*" and "*end*" are lines in Rit text. Extracted area is between them.

6. Examples

The use of Rit is almost trivial. Small examples listed below are limited to the ones that are useful to understand the behavior of Rit.

6.1. Command execution and newline control

You can write Rc script in Rc code area.
For example

```
Date: ${date}
```

will produce

```
Date: Thu Dec 23 10:17:10 JST 2004

```

Note that we have two subsequent *NL*s: one from “`date`” command and another from *NL* in the text. In case that we need to suppress *NL* from a command, we have “`}$`”:

```
${date} continues next line
${date}$ stays same line.
```

then the result will be:

```
Thu Dec 23 10:17:10 JST 2004
 continues next line
Thu Dec 23 10:17:10 JST 2004 stays same line.
```

6.2. Empty command

Be careful with the example:

```
${date;}$ is equivalent to ${date}.
```

will result in

```
Thu Dec 23 10:17:10 JST 2004
 is equivalent to Thu Dec 23 10:17:10 JST 2004
.
```

Why the *NL* in the output of “`date;`” is not suppressed? This is because the last command in “`date;`” is empty. Note that “`}$`” operates on the last command.

6.3. Embedded shell variables

Let “`alice`” be assigned to a variable “`user`”, and Rit text be

```
User: $user
This is equivalent to
User: ${echo -n $user}
```

then the above three lines are converted to:

```
User: alice
This is equivalent to
User: alice
```

6.4. Newline escape

A dollar at the end of line is NL escape. Example:

```
This line has NL escape. $
same line.
```

will be converted to:

```
This line has NL escape. same line.
```

Most Rc commands produce *NL* at the end. We can avoid redundant *NL* by putting “$” after “}” and/or before *NL*:

```
${pwd}$
this line will be next of pwd line.
${pwd}$$
this line stays in the same pwd line.
```

The result is

```
/usr/arisawa/src/pegasus-2.1/rit
this line will be next of pwd line.
/usr/arisawa/src/pegasus-2.1/ritthis line stays in the same pwd line.
```

6.5. Multi-lines in code areas

Rit has full functionality with Rc. For example Rit allows multi-line script:

```
${
book='Alice in Wonder Land'
}$
${
echo -n 'echo test of multi-line:
line1: Carol''s book:
line2: '$book'
line3: and we can use { and } in Rc strings'}
Back slash newline escape in Rc command will work:
${echo -n one \
two}
```

These lines will be converted to:

```
echo test of multi-line:
line1: Carol's book:
line2: Alice in Wonder Land
line3: and we can use { and } in Rc strings
Back slash newline escape in Rc command will work:
one two
```

But be careful with the following example:

```
${
echo alice
}$
will produce one empty line.
```

results in

```
alice

will produce one empty line.
```

because there exists an empty command at the left side of “}$”.

6.6. Comment in Rc code

Rc comment continues up to *NL* in accordance with Rc grammar. Therefore example

```
${# This is a comment up to NL } this is also a part of comment
# this is also a comment
} # not a comment
${# comment line1 terminated by Rc NL escape\
continued comment line
} # not a comment
$${# This isn't a comment but a part of text }
```

will produce

```
 # not a comment
 # not a comment
${# This isn't a comment but a part of text }
```

6.7. Sequence of dollars

If a sequence of two or more dollars appears, then one dollar is simply discarded. Thus "`$$$$home`" is converted to "`$$$home`" and "`$$$${not a Rc script}`" is converted to "`$$${not a Rc script}`". Dollars "`$$$$`" at the end of line is not a *NL* escape. That is just converted to "`$$$`".

6.8. Argument variable $0, $1, $2, ...

Among `$0`, `$1`, `$2`, `...`, variable "`$0`" is special. This is a file name currently processed. Remaining "`$1`", "`$2`", `...` are arguments.

For example, let a file "`foo`" be

```
$0
$1
$2
```

then we have

```
term% rit foo alice bob
foo
alice
bob
term%
```

6.9. Rit executable

In most cases, Rit will be used in executable file. Then the meaning "`$0`", "`$1`", `...` are shown by the following example:

```
term% cat>bar
#!/bin/rit -s
$0
$1
$2
term% chmod 755 bar
term% bar alice bob
./bar
alice
bob
term%
```

For the case we want only base name of “$0”, we have “-b” option:

```
term% cat>bar
#!/bin/rit -bs
$0
$1
$2
term% chmod 755 bar
term% bar alice bob
bar
alice
bob
term%
```

6.10. Termination

Rc block

```
${echo exit 'some message'>[1=2];exit}
```

will terminate Rit. Rc function “quit” is predefined in Rit:

```
fn quit {echo exit $1 >[1=2];exit}
```

Simple “exit” does not terminate Rit but next ${ } block will terminate Rit because of error. For example

```
${exit} ${}
```

will terminate Rit.

Example 1:

```
term% rit
${quit}
term% echo $status

term%
```

Example 2:

```
term% rit
${quit abcd}
term% echo $status
rit 619: abcd
term%
```

where the number 619 is process ID.

7. Known Bugs

7.1. Echo -n

Use of "/bin/echo -n" can make a problem which comes from "0 byte write problem to a pipe": let "foo" be any executable, then it is not guaranteed that the following two commands

```
foo > bar
```

and

```
foo | cat > bar
```

produce same "bar".

Avoiding this problem, Rc function "echo" is predefined internally as

```
fn echo {
        if(~ $1 '-n'){
                shift
                if(~ $"* ?*)/bin/echo -n $*
        }
        if not /bin/echo $*
}
```

7.2. Last command with no output

Due to implementation of Rit

```
${name=alice}$ $name
```

does not produce "alice". Assignment just before "}$" is not only no effect but also can terminate Rit in certain conditions. Write instead

```
${name=alice} $name
```

or

```
${name=alice;}$ $name
```

Assignment is merely an example. Except for empty command, any command with no output has same problem.

7.3. Here document of Rc

Here document of Rc might make problem for Rit. However here document will not be used in Rit, because: (1) Rc does not handle here documents properly inside of Rc's "{ }" blocks; (2) on the other hand, here documents outside of these blocks can be moved to text area of Rit.

8. Discussion

8.1. Backward operation

Backward operation "}$" is powerful. The operation will be required as long as Rit claims to be "*general-purpose*". However backward operation makes things complicated. As far as HTML is concerned, "}$" is over-specification, because HTML is insensitive to *NL*.

As a matter of fact, I have no experience of writing Web pages using "${ ... }" nor "${ ... }$" except in the syntax:

```
${
...
}$
```

where "..." is Rc code. In this syntax, "}$" can be replaced by "}" but I prefer "}$" because of clarity.

8.2. Porting to UNIX

Clean porting of Rit to UNIX might be difficult because

- named pipe of UNIX is shared pipe and therefore it is impossible to protect the pipe from other processes,
- the pipe must be explicitly unlinked which means the name will be left by an accidental death of Rit.

8.3. Security

Rc code area in Rit text must be carefully coded in case that Rit is applied to Web application, because the area is essentially CGI. All cautions[4] on CGI are applied as well to Rit. CGI of Rit is based on Rc with which we can derive all the power of Plan 9 tools. In other words, It is desirable for Rit to be used under the Web server such as Plan9/Pegasus[5] that is running with weak privilege and in confined name space.

8.4. Live examples

Apart from personal Web pages and unpublished applications, Remoty[6] — a home page management tool that runs on Plan9/Pegasus — is the only example written in Rit. Remoty consists of 1127 lines. Since Remoty has text editor written in Javascript, nearly half of the lines are occupied by Javascript code. The example shows some techniques to construct medium size applications and also shows how to invoke CGI script using flexible CGI handler of Pegasus.

9. References

[1] Kenji Arisawa, Rit source code
http://plan9.aichi-u.ac.jp/netlib/cmd/rit/

[2] Tom Duff. "Rc — The Plan 9 Shell", *Plan 9 - The document - Volume Two*, 1995
http://plan9.bell-labs.com/sys/doc/rc.html

[3] Rc source code
http://cm.bell-labs.com/sources/plan9/sys/src/cmd/rc/

[4] The World Wide Web Security FAQ
http://www.w3.org/Security/Faq/wwwsf4.html

[5] Kenji Arisawa, “Pegasus Project”
http://plan9.aichi-u.ac.jp/pegasus/

[6] Kenji Arisawa, “Remoty”
http://plan9.aichi-u.ac.jp/pegasus/appls/remoty/index.html